2.2 3D Capabilities

While the API or software application takes care of the geometry and lighting stages of the 3D pipeline, the Intel740™ graphics accelerator enables hardware acceleration of the rendering stages. In the DirectX and OpenGL 3D Pipeline diagrams (Figure 2-7 and Figure 2-8), the rasterization stage of the 3D pipeline consists of the Setup EngineScan Converter,Texture Pipeline, and Color Calculator Depth Buffer Test. These four modules comprise the rendering engine and this section discusses all of the rendering features associated with the 3D hardware including the following subsections for both OpenGL and DirectX:

2.2.1 3D Pipeline

The 3D pipeline unit in the Intel740™ graphics accelerator offers advantages over the traditional graphics accelerators by performing 3D setup locally rather than within the CPU. This difference allows the processor to perform more geometry calculations while the Intel740™ graphics accelerator performs set-up and rendering. 3D features supported include perspective correct texture mapping, trilinear mipmapping, Gouraud shading, alpha-blending, stippling, and Z-buffering. Depending on the application, each feature can be independently enabled or disabled for various levels of performance. The Intel740™ graphics accelerator allows for high performance when all 3D features are enabled for the entire run of the application with the only exception being antialiasing. The Intel740™ graphics accelerator is optimized for high throughput when textures are stored in AGP memory, otherwise known as non-local video memory. Relocating textures in main memory is also supported. Locating texture information in the AGP non-local video memory frees up the Intel740™ graphics accelerator local frame buffer memory for graphics execution. Textures cannot be put in local video memory. Polygon antialiasing is hardware assisted by Intel740™ graphics accelerator.

Figure 2-7 and Figure 2-8 illustrates the DirectX and OpenGL API function calls, respectively, as they are used in the 3D rasterization pipeline of the Intel740™ graphics accelerator architecture.

Figure 2-7. 3D Pipeline for DirectX

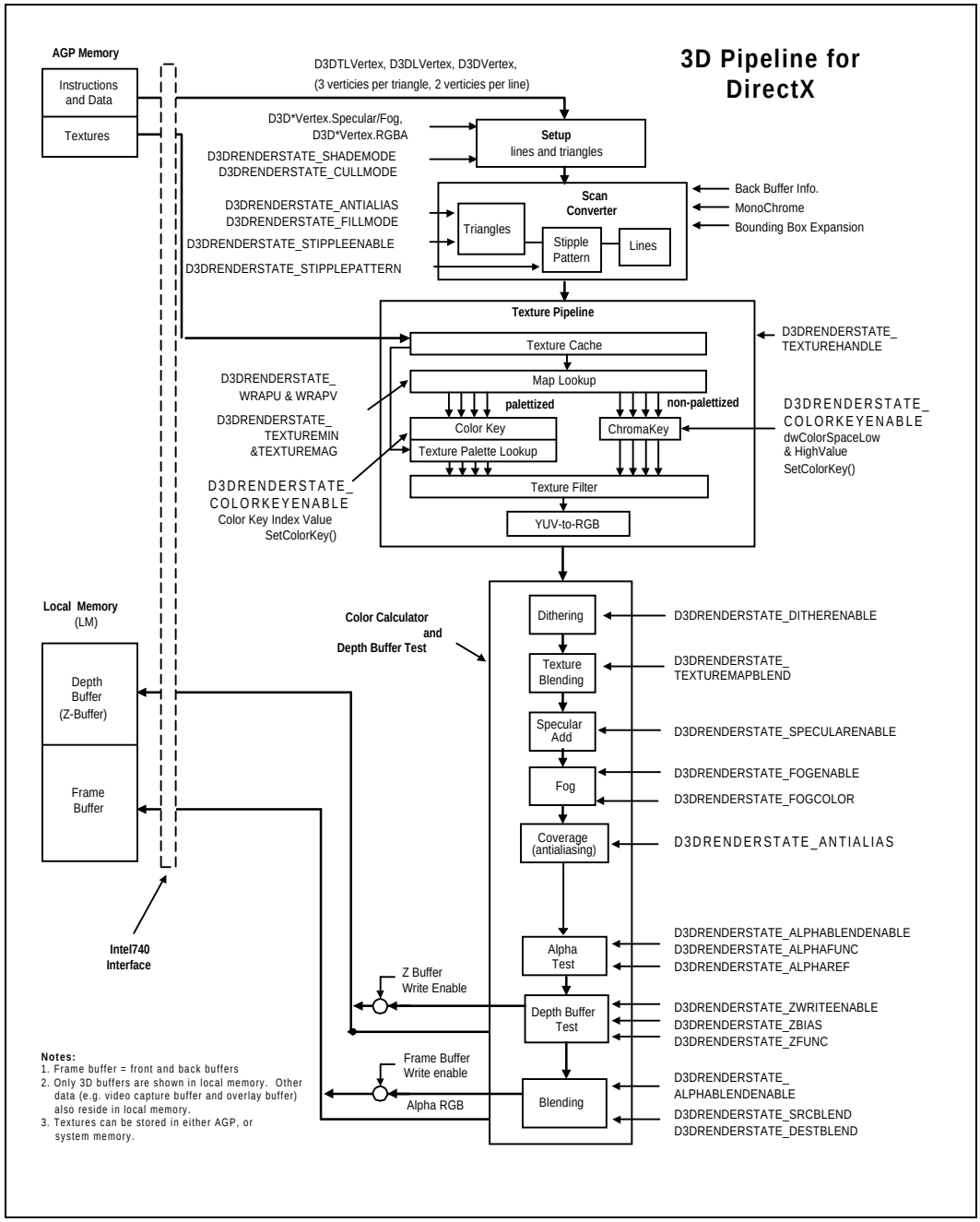
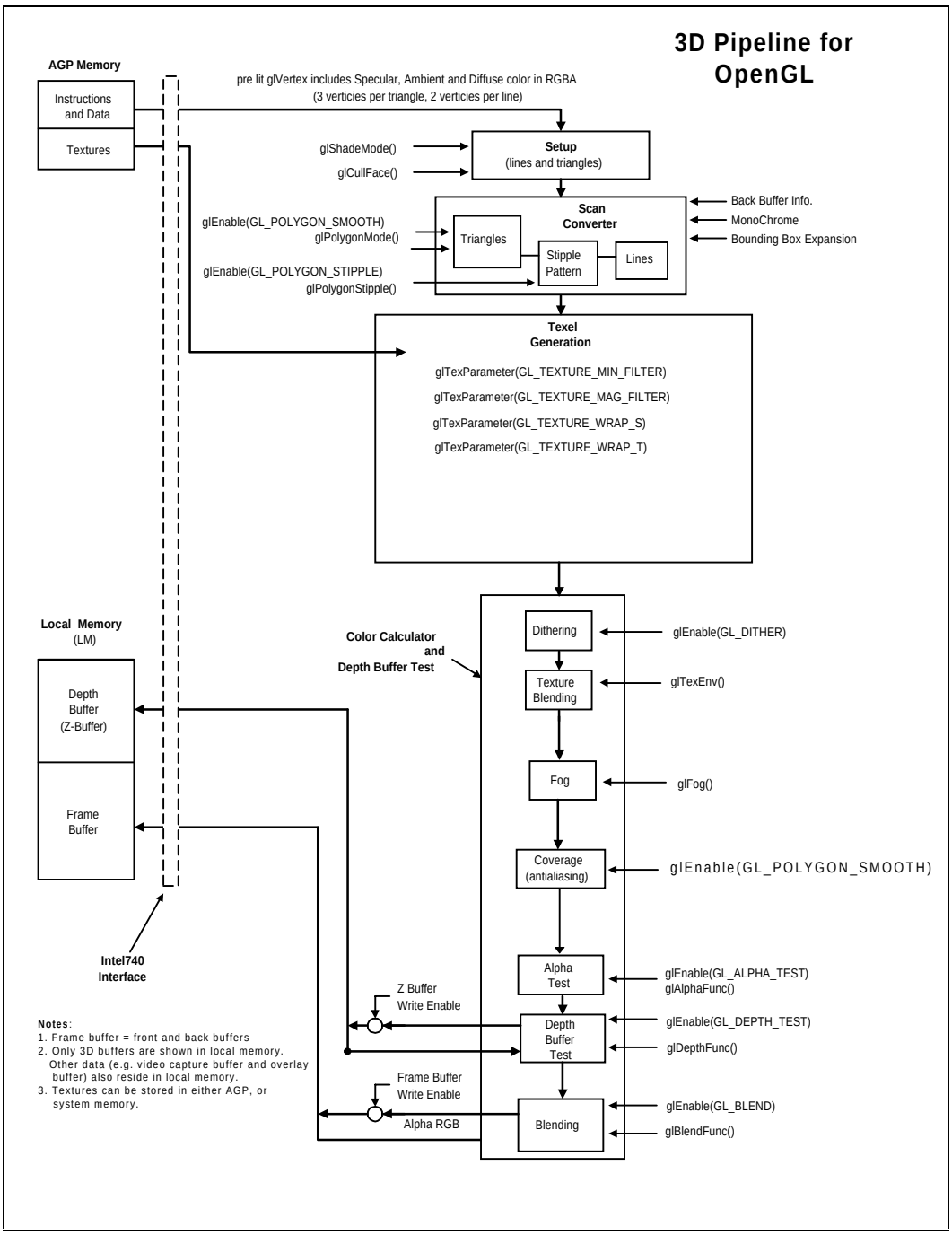


Figure 2-8. 3D Pipeline for OpenGL



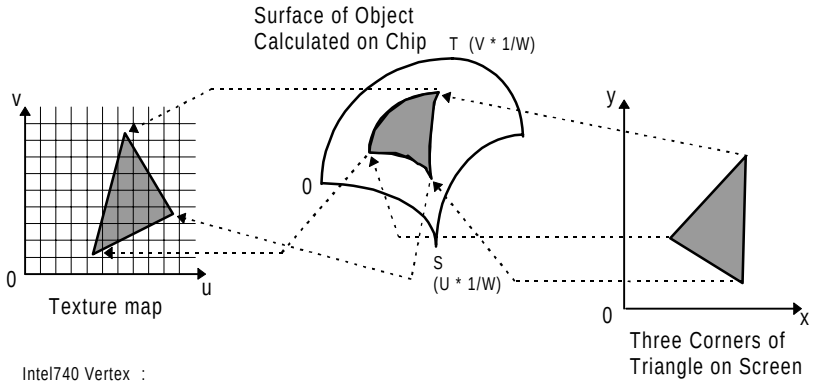
The four main modules within the 3D Pipeline are:

Setup Engine	The Setup Engine performs the necessary calculations to make the geometry data useful for the rest of the pipeline. Some of the functions include culling, and perspective correct calculation of texture coordinates as they correspond to pieces of the geometry.
Scan Converter	The Scan Converter performs functions in parallel with the Setup Engine to read vital information such as fog, specular RGB, and blending data and sends it on to the Texture Pipeline so that the Texture Pipeline does not have to stop the flow of the pipeline in order to wait for this data.
Texture Pipeline	The Texture Pipeline receives the texture coordinate data information from the Setup Engine and texture blend information from the Scan Converter and stores this information in the texture cache. It performs texture chroma and color key match, texture bilinear interpolation, and YUV to RGB conversions.
Color Calc./Depth Test	The Color Calculator/Depth Test is where the color data such as fogging, specular RGB, texture blend, and alpha blend is processed. The Color Calculator computes the resulting color of a pixel. The red, green, blue, and alpha are combined with the corresponding components resulting from the Texture Pipeline unit. These textured pixels are then modified by the specular and fog parameters to create specular highlighted, fogged, textured pixels which are color blended with the existing values in the frame buffer. Alpha and depth buffer tests are conducted which will determine whether the frame and depth buffers will be updated with new pixel values.

2.2.2 3D Primitives

The 3D primitives are lines, triangles, and state variables. Pipeline flushes occur when updating the palette and stipple memories, since these are too large to allow pipelining of their data. In either case, all primitives rendered after a change in state variables will reflect the new state. Figure 2-9 shows the triangle data structure which is handled by the Intel740™ graphics accelerator drivers and also shows how the texture is mapped from the texture coordinate U, V space to the normalized S, T object space where perspective correction is applied to the texture as well as simulated curvature before being mapped to the object in X, Y screen coordinates. The triangle data structure is passed to the Intel740™ graphics accelerator drivers by either the DirectX or the OpenGL API call backs.

Figure 2-9. Triangle as the Intel740™ Graphics Accelerator Driver Sees It




```
Intel740 Vertex  :

struct {
        float       X;                          /* 0.0  -  2047.0  */
        float       Y;                          /* 0.0  -  1023.0  */
        float       Z;                          /* 0.0  -  1.0,  0 - 64K  */
        float       W;                          /* 1/Z  */
        struct {
                unsigned char       blue;       /*  0  -  255  */
                unsigned char       green;      /*  0  -  255  */
                unsigned char       red;        /*  0  -  255  */
                unsigned char       alpha;      /*  0  -  255  */
        }  dwColor;
        struct {
                unsigned char       sblue;      /*  0  -  255  */
                unsigned char       sgreen;     /*  0  -  255  */
                unsigned char       sred;       /*  0  -  255  */
                unsigned char       fog;        /*  0  -  255  */
        }  dwSpecularColor;
        float       U;                          /*  S15.16 0 - 64K */
        float       V;                          /*  S15.16 0 - 64K */
} Triangle[3];
```

Example 2-1. Sending Data to the Intel740™ Graphics Accelerator Using DirectX

When using DirectX, the data format for a vertex which can be sent to the Intel740™ graphics accelerator driver via a DirectX execute buffer, or by using the DrawPrimitive or DrawIndexedPrimitive command is a D3DTLVERTEX, D3DLVERTEX, or D3DVERTEX data structure. The Intel740™ graphics accelerator does the rasterization or rendering portion of the 3D pipe. The user must set up the appropriate lighting and transforms regardless of vertex type. The difference is that the DirectX API will know to perform lighting and transforms as preset by the user when a D3DVERTEX is sent, or just transforms when the D3DLVERTEX is sent. Lighting and transformation is not done by the Intel740™ graphics accelerator, but will be done by the API software in these instances. See the Microsoft DirectX 5.0 documentation for more information on how to set up the lighting and transformations. The D3DTLVERTEX data structure is illustrated below.

```
D3DTLVERTEX TYPE
 typedef struct _D3DTLVERTEX {
    union {
        D3DVALUE sx; // sx is the screen coordinate of the x position of the vertex
        D3DVALUE dvSX;
    };
    union {
        D3DVALUE sy  // sy is the screen coordinate of the y position of the vertex
        D3DVALUE dvSY;
    };
    union {
        D3DVALUE sz; // sz is the z position of the vertex used for z compares
        D3DVALUE dvSZ;
    };
    union {
        D3DVALUE rhw;// rhw is the 1/z value for the vertex or the reciprocal
                     //of homogeneous
        D3DVALUE dvRHw;// w. This value is 1 divided by the distance from the
                     //origin to the object
// along the z-axis.
    };
    union {
        D3DCOLOR color; // color corresponds to the vertex color components of red,
                        //green, blue, and alpha.
D3DCOLOR dcColor;
    };
    union {
        D3DCOLOR specular; // specular corresponds to the vertex specular color
                           //component
        D3DCOLOR dcSpecular; // consisting of sred, sgreen, and sblue.  The alpha of
                        //the specular color is used for  the fog density value.
    };
    union {
        D3DVALUE tu; //  tu corresponds to the texture map horizontal component.
        D3DVALUE dvTU;
    };
    union {
        D3DVALUE tv; // tv corresponds to the texture map vertical component.
        D3DVALUE dvTV;
    };
} D3DTLVERTEX, *LPD3DTLVERTEX;
```

The Intel740™ graphics accelerator supports the following different D3DPRIMITIVETYPEs for DrawPrimitive:

D3DPT_POINTLIST	Renders a collection of isolated points
D3D_LINELIST	Renders a list of isolated straight line segments
D3DPT_LINESTRIP	Renders a single polyline
D3DPT_TRIANGLELIST	Renders a sequence of isolated triangles
D3DPT_TRIANGLESTRIP	Renders a triangle strip
D3DPT_TRIANGLEFAN	Renders a triangle fan

Below is the DirectX function prototype for DrawIndexPrimitive which is used to call the Intel740™ graphics accelerator driver to take the triangle data and begin the hardware rasterization process.

```
HRESULT IDirect3DDevice2::DrawIndexedPrimitive(
D3DPRIMITIVETYPE type,
D3DTLVERTEXTYPE D3DTLVertex,
LPVOID  VertexsListPointer,
DWORD VertexsCount,
LPWORD VertexsIndexList,
DWORD VertexsIndexCount,
DWORD DrawIndexedPrimitiveFlags);
```

The following code segment illustrates using DrawIndexPrimitive to send the vertex data to the Intel740™ graphics accelerator, assuming that the triangle information is ready for rendering:

```
HRESULT ddval
LPDIRECT3DDEVICE lpDev;
TransformVerticesTo3DView();
LightVertices();
TransformVerticesTo2DScreen();
if ((ddrval = lpDev->BeginScene()) != D3D_OK)
    return FALSE;
//begining of atomic block for Direct 3D rendering
ddrval=lpDev->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                          D3DVT_TLVERTEX,
                          (LPVOID)pvTLVertex,
                          iNumVertex,
                          (LPWORD)pdwIndex,
                          iNumFaces*3,
                          0) ;
if (ddrval != DD_OK)
    return FALSE;
//end of atomic block for Direct 3D rendering
if ((ddrval = lpDev->EndScene()) != D3D_OK)
    return FALSE;
```

It is best to do the transformations and lighting for the entire scene before the rendering, as implied in the code segment above. Multiple triangle lists can be sent within the BeginScene() and EndScene() call without hampering the performance. A triangle list larger than 85 triangles is recommended while a list of 512 triangles is optimal. See Chapter 4 for in-depth triangle list performance information.

Example 2-2. Sending Data to the Intel740™ graphics accelerator Using OpenGL

The three ways to send rendering information to the Intel740™ graphics accelerator using OpenGL are immediate method, vertex arrays, and display lists. This document first shows the immediate method, which is straightforward and which helps to understand the second and preferred vertex array method. The display list method is not discussed in this document; it can be found in the *OpenGL Programming Guide*. This document is concerned with showing the user how to implement OpenGL calls which will utilize the features of the Intel740™ graphics accelerator; therefore, this manual will not discuss overall OpenGL programming methods. It should be noted that the OpenGL vertex information sent to the Intel740™ graphics accelerator will be pre-lit, which means that the RGBA component will have already included the specular, diffuse and ambient lighting for the vertex.

OpenGL describes vertex information a little bit differently than DirectX. For instance, to specify an OpenGL vertex and its surface and texture attributes the following code could be used:

```
glBegin();
    glColor*();// Set current color
    glTexCoor*();// Set texture coordinates
    glEdgeFlag*();// Control drawing of edges
    glVertex*();// Set vertex coordinates
glEnd();
```

"*" specifies the type of arguments the function call will pass in the function parameters. For glVertex, the types conform to the following:

```
void glVertex{234}{sifd}[v](TYPE coords);
```

Where "(234)" specifies the number of coordinates from as few as two for (x,y) to as many as four for (x,y,z,w). Then the "{sifd}" portion describes the data type as either "short", "int", "float", or "double." The next portion of the function, "{v}" is used to specify that a pointer to a vector (or array) will be past in the parameter rather than a series of individual arguments.

It is important to send the glVertex() command last, because the information sent previously will be used to describe the vertex at this point.

To describe all of the component information of a vertex including the texture coordinates, color, and edge flags, each of the functions between the glBegin() and glEnd() may be called. Before making the glColor call, other calls to set the specular lighting, fogging and antialiasing methods should be called. These calls are discussed in the 3D features section of this document where for each feature of the Intel740™ graphics accelerator such as fogging, an OpenGL implementation is provided. The glBegin() and glEnd() are used to specify the beginning and end of an atomic primitive. There are different types of primitives which can be passed as arguments to glBegin(). They are as follows:

GL_POINTS	Renders a collection of isolated points
GL_LINES	Renders a list of isolated straight line segments
GL_TRIANGLES	Renders a sequence of isolated triangles
GL_LINE_STRIP	Renders a single polyline
GL_TRIANGLE_STRIP	Renders a triangle strip
GL_TRIANGLE_FAN	Renders a triangle fan
GL_QUAD	Renders a quad triangulated into individual triangles
GL_QUAD_STRIP	Renders quadrilateral strips triangulated into individual triangles
GL_POLYGON	Renders polygons triangulated into individual triangles

When using OpenGL, the best way to send vertex data to the driver is to use vertex arrays, which minimize the number of function calls required for one geometric object. Vertex arrays are a new feature of OpenGL 1.1. For the Intel740™ graphics accelerator, it is best to minimize these function calls to improve performance and to reduce the redundant processing of shared vertices. The way to use the vertex arrays is as follows:

1. Enable each array type to be used:

   ```
   void glEnableClientState(Glenum array);
   ```

   Where array is one of the following symbolic constants: GL_VERTEX_ARRAY, GL_COLOR_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_TEXTURE_COORD_ARRAY, GL_EDGE_FLAG_ARRAY.

2. Point to each array to be rendered:

   ```
   void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid
   *pointer);
   void glTexCoordPointer(GLint size, GLenum type, GLsizei stride, const GLvoid
   *pointer);
   void glEdgeFlAGPointer(GLsizei stride, const GLvoid *pointer);
   void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid
   *pointer);
   ```

   GLint size: is the number of coordinates per vertex, which must be 2, 3, or 4.

   GLenum type:is the data type (GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE).

   GLsizei stride:is the byte offset between consecutive vertices (or other type).

   GLvoid *pointer:points to the storage array for the vertices (or other type).

Note: There is such a thing as “intertwined” arrays where multiple types can be stored in a single array and, therefore, can be “pointed to” using the stride variable to indicate the offset from the beginning of the first group to the beginning of the next group of the type to be pointed to. For example, an intertwined array of x, y, z vertices and RGB color could be created and pointed to in this way:

```
static GLfloat  intertwinded[] =
{2.0, 0.3, 2.0, 200.0, 100.0, 0.0,
2.0, 0.3, 0.0, 100.0, 100.0, 0.0,
2.0, 2.0, 0.3, 100.0, 300.0, 0.0};
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glColorPointer(3, GL_FLOAT,  6 * sizeof(GLfloat), intertwined);
glVertexPointer(3, GL_FLOAT, 6 * sizeof(GLfloat), &intertwined[3]);
```

3. Render the data. The above calls remain on the application side of the graphics pipeline. In order to send the data to the Intel740™ graphics accelerator for rendering the user needs to “dereference” the arrays which cause them to be sent down the graphics processing pipeline. This can be done by either de-referencing a single array element from a sequence of array elements or from an ordered list of array elements. The following call is used to render an ordered list of array elements:

   ```
   void glDrawArrays(GLenum mode, GLint first, GLsizei count);
   ```

   GLenum mode: The primitive type.

   GLint first: The start of the array to be processed

   GLsizei count: The number of elements to be rendered.

   ```
   glEnableClientStat(GL_COLOR_ARRAY);
   glEnableClientState(GL_VERTEX_ARRAY);
   .
   .
   glEnableClientState(otherarray);
   glColorPointer(3, GL_FLOAT, 6 * sizeof(GLfloat), intertwined);
   glVertexPointer(3, GL_FLOAT, 6 * sizeof(GLfloat), &intertwined[3]);
   .
   .
   ```

```
gl*Pointer(...);
glDrawArrays(GL_TRIANGLES, 0, vertexs_count);
glDisableClientState(Glenum array);
```

The above call would render all of the arrays which have been enabled and pointed to.

2.2.3 Data Formats

The data value ranges are independent of the API. Table 2-1 lists each data format and the corresponding domain and range values.

Table 2-1. Data Formats

Parameters	Input Format	Domain	Range
Vertex X, Y	32-bit Floating Point	0.0–2048	x: 0–2047
y: 0–1023	Depth (Z)	32-bit Floating Point	0.0–1.0
0–64K	Texture U, V	32-bit Floating Point	0–64K
0–64K (32K)	Texture W	32-bit Floating Point	0.0–1.0
1/z	Color R, G, B, A	Fixed 0.8	0–255
0–255	Specular Color R, G, B	Fixed 0.8	0–255
0–255	Fog Factor	Fixed 0.8	0–255
0–255			

2.2.4 Surface Color Attributes

Surface attributes are those items which allow the user to define the object's visual quality and which can be combined in a number of ways to create different atmospheric and lighting effects. The surface attributes which the Intel740™ graphics accelerator supports are discussed in the following subsections:

- "Fogging" (below)
- "Specular Highlighting" on page 2-19
- "Alpha Testing" on page 2-23
- "Color Dithering" on page 2-23
- "Shading" on page 2-24
- "Stippled Pattern" on page 2-25

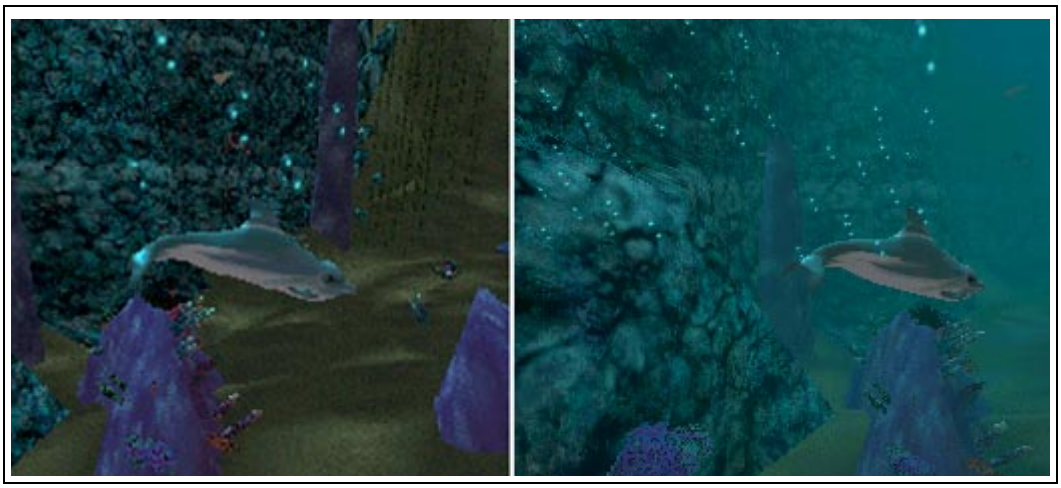
2.2.4.1 Fogging

Fogging adds the effect of density to the atmosphere. As an object goes farther away from the viewer, it appears to become more "cloudy" or "foggy" than closer objects. Fogging is specified at each vertex and is interpolated to each pixel center. If fog is disabled, the incoming color intensities are passed unchanged. Fog is linearly interpolative, with the pixel color determined by the following equation:

$$C = f * Cp + (1 - f) * Cf$$

where f is the fog coefficient per pixel, Cp is the polygon color, and Cf is the fog color.

Figure 2-10. Effects of Fogging Off vs Fogging On



Example 2-3. Enabling Fogging with DirectX

The following code shows how to enable fogging using the DirectX API. The first step is to turn fogging on by setting the "D3DRENDERSTATE_FOGENABLE" state to "TRUE". The second step is to set the color of the fog as shown below where D3DCOLOR has a red, green and blue value that will correspond to the color of the fog.

```
SetRenderState(D3DRENDERSTATE_FOGENABLE, TRUE);
SetRenderState(D3DRENDERSTATE_FOGCOLOR, <D3DCOLOR>);
```

The density of the fog is specified by setting the alpha component of the specular value of a vertex as shown below using the D3DLVERTEX data type:

```
D3DLVERTEX pLVertex;
pLVertex.specular = RGBA_MAKE( sred, sgreen, sblue, FOG_DENSITY);
```

The density of the fog value is between 0 and 255, where 0 is dense, completely opaque fog and 255 completely clear or no fog.

Example 2-4. Enabling Fogging with OpenGL

There are several steps and many choices when implementing fogging through the OpenGL API. The following code shows how to set the multiple fogging values:

```
glEnable(GL_FOG) {  ...  };
```

Enables fogging; other values corresponding to the fog can be set within the braces.

```
glFogi(GL_FOG_MODE, <MODE>);
```

Where <MODE> is either GL_LINEAR, GL_EXP, or GL_EXP2. The GL_LINEAR flag is hardware accelerated with the Intel740™ graphics accelerator.

```
GLfloatfogColor[4] = {0.5, 0.5, 0.5, 1.0};
glFogfv(GL_FOG_COLOR, fogColor);
```

Sets the fog color from the values set in the fogColor array. Fog color can be set as RGB values or from a color index.

```
glFogf(GL_FOG_DENSITY, <VALUE>);
```

Sets the fog density to <VALUE> which can be a floating point number from 0.0 to 1.0. The fog density is used when calculating GL_EXP or GL_EXP2 fog values.

```
glFogf(GL_FOG_START, <START_VALUE>);
```

Sets the start of the fog in the view. The <START_VALUE> corresponds to a “z” value in the view and can be any floating point value within the view volume z range.

```
glFogf(GL_FOG_END, <END_VALUE>);
```

Sets the end of the fog in the view. The <END_VALUE> corresponds to the point in the view where the user wants fogging to end and can be a floating point value with the view volume z range.

```
glHint(GL_FOG_HINT, <HINT_VALUE>);
```

Specifies how the fog is calculated where <HINT_VALUE> is either GL_NICEST or calculated per pixel, or GL_FASTEST, calculated per vertex. The Intel740™ graphics accelerator accelerates GL_FASTEST.

For OpenGL, the fog equations are as follows:

$f = e^{-(density * z)}$ (GL_EXP)

$f = e^{-(density*z)2}$ (GL_EXP2)

f = end - z/end - start (GL_LINEAR)

2.2.4.2 Specular Highlighting

Specular highlighting adds the effect of a “hot spot” on an object which corresponds to the shininess of the material. The specular highlight can be varied by the amount specified for each red, green, and blue component. The Intel740™ graphics accelerator has the capability to utilize colored specular highlights which adds to the realism of a scene. For instance, if you have a blue light shining on a red apple, the specular highlight would be blue in real life. With the Intel740™ graphics accelerator, it is possible to create a specular highlight of any color.

Figure 2-11. Effects of Using Specular Highlighting

Example 2-5. Enabling Specular Highlighting with DirectX

The specular color of a vertex is set to red as illustrated with the following DirectX code:

```
D3DLVERTEX pLVertex;
pLVertex.specular = RGBA_MAKE( 255, 0, 0, FOG_DENSITY);
```

In order to enable the specular highlights with DirectX so that they are visible, the following D3DRENDERSTATE is set to true:

```
SetRenderState(D3DRENDERSTATE_SPECULARENABLE, TRUE);
```

Example 2-6. Enabling Specular Highlighting with OpenGL

Specular highlighting is added in to the color equation at the application's lighting stage which formulates the RGBA color sent to the driver. To set the specular lighting component in OpenGL the following code may be used:

```
Glfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0}
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
```

2.2.4.3 Alpha Blending

Alpha blending adds the material property of transparency or opacity to an object. Alpha blending requires a source red, green, blue, and alpha component and a destination red, green, blue and alpha component. Using these two components, for example, a glass surface on top (source) of a red surface (destination) would allow much of the red base color to show through. The Intel740™ graphics accelerator blends the source Rs, Gs, Bs, As component with the destination Rd, Gd, Bd, Ad component by the following formula:

$$(R', G', B', A') = (RsSr + RdDr, GsSg + GdDg, BsSb + BdDb, AsSa + AdDa)$$

Where Sr, Sg, Sb, Sa is a blending factor for the source and Dr, Dg, Db, Da is a blending factor for the destination.

Figure 2-12. Effects of Using Alpha Blending

Example 2-7. Enabling Alpha Blending with DirectX

To enable alpha blending with DirectX, the ALPHABLENDENABLE flag must be set to TRUE, and then a SRCBLEND and DESTBLEND flag must be specified as shown below:

```
SetRenderState(D3DRENDERSTATE_ALPHABLENDENABLE, TRUE);
SetRenderState(D3DRENDERSTATE_SRCBLEND, <D3DBLEND FLAG>);
SetRenderState(D3DRENDERSTATE_DESTBLEND, <D3DBLEND FLAG>);
```

The D3DBLEND FLAG is ZERO, ONE, SRCCOLOR, INVSRCCOLOR, DESTCOLOR, INVDESTCOLOR, BOTHSRCALPHA, or BOTHINVSRCALPHA. The blending factors are calculated depending on the D3DBLEND FLAG according to the formulas shown in Table 2-2. A common implementation is to set the source flag to SRCCOLOR and the destination flag to INVSRCCOLOR.

Example 2-8. Enabling Alpha Blending with OpenGL

To enable alpha blending with OpenGL, the following function call is made:

```
glEnable(GL_BLEND);
```

To set the source and destination blending factors, the following call is made:

```
glBlendFunc(<SOURCE_FLAG>, <DESTINATION_FLAG>);
```

The <SOURCE_FLAG> and <DEST_FLAG> can be set to any of the flags in the chart below and the resulting blend will be what the corresponding values equate to when plugged into the Intel740™ graphics accelerator equation above.

Table 2-2. Alpha Blend Functions for OpenGL & DirectX

FLAG	Source Blend Factor	Destination Blend Factor
GL_ZERO D3DBLEND_ZERO	Sr = 0 Sg = 0 Sb = 0 Sa = 0	Dr = 0 Dg = 0 Db = 0 Da = 0
GL_ONE D3DBLEND_ONE	Sr = 1 Sg = 1 Sb = 1 Sa = 1	Dr = 1 Dg = 1 Db = 1 Da = 1
GL_SRC_COLOR D3DBLEND_SRCCOLOR	Sr = Rs Sg = Gs Sb = Bs Sa = As	
GL_DST_COLOR D3DBLEND_DESTCOLOR		Dr = Rd Dg = Gd Db = Bd Da = Ad
GL_ONE_MINUS_SRC_COLOR D3DBLEND_INVSRCCOLOR	Sr = 1-Rs Sg = 1-Gs Sb = 1-Bs Sa = 1-As	
GL_ONE_MINUS_DST_COLOR D3DBLEND_INVDESTCOLOR		Dr = 1-Rd Dg = 1-Gd Db = 1-Bd Da = 1-Ad
GL_SRC_ALPHA D3DBLEND_SRCALPHA	Sr = As Sg = As Sb = As Sa = As	Dr = As Dg = As Db = As Da = As
GL_ONE_MINUS_SRC_ALPHA D3DBLEND_INVSRCALPHA	Sr = 1-As Sg = 1-As Sb = 1-As Sa = 1-As	Dr = 1-As Dg = 1-As Db = 1-As Da = 1-As
D3DBLEND_BOTHSRCALPHA	Sr = As Sg = As Sb = As Sa = As	Dr = 1-As Dg = 1-As Db = 1-As Da = 1-As
D3DBLEND_BOTHINVSRCALPHA	Sr = 1-As Sg = 1-As Sb = 1-As Sa = 1-As	Dr = As Dg = As Db = As Da = As

2.2.4.4 Alpha Testing

The Intel740™ graphics accelerator supports the use of alpha blend testing functions. This allows the user to control how objects in the scene are alpha blended. When using source alpha blending, the user does not need to create an alpha buffer. When using source alpha blending, the alpha channel of the textures are used for the blending formulas and there is no need for an alpha buffer. The user must remember to sort from back to front, so that the blending is performed correctly.

Example 2-9. Enabling Alpha Testing Functions With DirectX

To enable alpha testing functions with DirectX, the following render states are set:

```
SetRenderState(D3DRENDERSTATE_ALPHABLENDENABLE, TRUE);
SetRenderState(D3DRENDERSTATE_ALPHATESTENABLE, TRUE);
SetRenderState(D3DRENDERSTATE_ALPHAFUNC, <D3DCMPFUNC>);
SetRenderState(D3DRENDERSTATE_ALPHAREF, <ALPHA REF> );
```

Where <D3DCMPFUNC> can be set to D3DCMP_NEVER, D3DCMP_LESS, D3DCMP_EQUAL, D3DCMP_LESSEQUAL, D3DCMP_GREATER, D3DCMP_NOTEQUAL, D3DCMP_GREATEREQUAL, or D3DCMP_ALWAYS. And where <ALPHA REF> is a value specifying a reference alpha value against which pixels are tested when alpha-testing is enabled. This value is in the range of 0 to 1 and must be 8 bits or less for the Intel740™ graphics accelerator. The default value is 0.

Example 2-10. Enabling Alpha Testing Functions With OpenGL

To enable alpha testing functions with OpenGL, the following render states are set:

```
glEnable(GL_ALPHA_TEST);
glAlphaFunc(<GLFUNC>, <GLREF>);
```

Where <GLFUNC> is GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or GL_NOTEQUAL. <GLREF> must be between 0 and 1.

2.2.4.5 Color Dithering

Color dithering is created by a pattern of pixels which are more than one color. When looked at from a distance, the combined effect is a new color. In this manner, many different colors can be simulated by combining a few colors. Color dithering takes advantage of the human eye's propensity to "average" the colors in a small area. With limited color fidelity available, large areas of "flat" colors can exist. Color dithering takes the input of color, alpha, and fog components and converts them from 8 bits to five- or six-bit components. Color dithering simulates 256-level color resolution by an ordered pattern of 32- or 64-level color pixels. A four-bit dither value is obtained by addressing a 4x4 matrix with the pixel's x and y (2 LSBs of each). The matrix repeats every four pixels in both directions. The value obtained is appropriately shifted to align with (what would be otherwise) truncated bits of the component being dithered. It is then added with the component and the result is truncated to the five (six for green) MSBs.

Example 2-11. Enabling Color Dithering with DirectX

To enable color dithering with DirectX do the following:

```
SetRenderState(D3DRENDERSTATE_DITHERENABLE, TRUE);
```

Example 2-12. Enabling Color Dithering with OpenGL

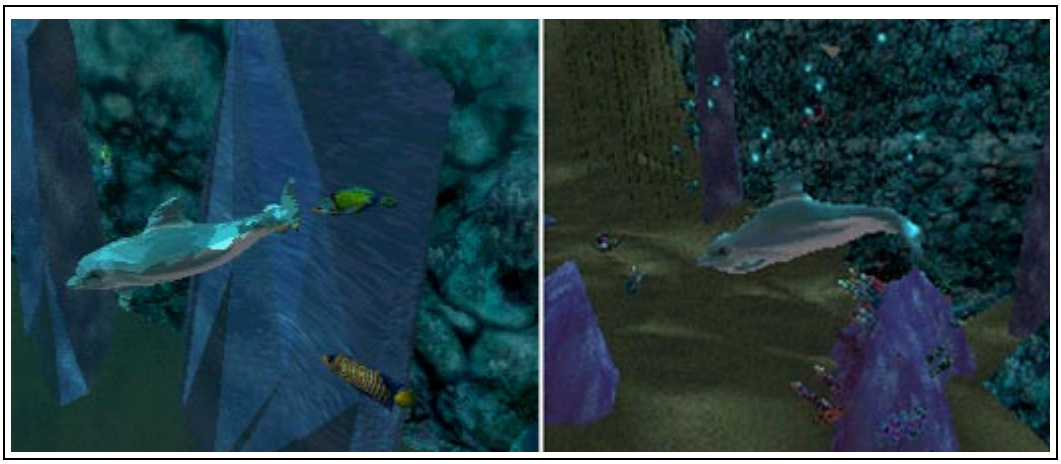
To enable color dithering with OpenGL do the following:

```
glEnable(GL_DITHER);
```

2.2.4.6 Shading

The Intel740™ graphics accelerator shading attributes determine how the colors of the polygons (triangles) are interpolated for each pixel in a surface. The Intel740™ graphics accelerator allows each of the alpha, fog, specular, and color attributes to be shaded individually. There are two types of shading performed by the Intel740™ graphics accelerator: flat shading and Gouraud shading. Flat shading makes objects appear blocky, since each polygon (triangle) face is denoted by a solid color. This is because flat shading takes a specified attribute from the first passed vertex and uses this attribute to cover every pixel in the polygon. Gouraud shading smooths the appearance of adjacent polygons (triangles) so that a sphere which looked blocky flat shaded can be made to look more rounded. This is because Gouraud shading takes the three vertices of the triangle and interpolates over the entire surface to blend the vertex colors and attributes such as fog, specularity and transparency (alpha).

Figure 2-13. Effects of Flat Shading vs. Gouraud Shading



Example 2-13. Enabling Shading with DirectX

To enable either flat or Gouraud shading using DirectX, the following render state is set:

```
SetRenderState(D3DRENDERSTATE_SHADEMODE, <D3DSHADEMODE>);
```

Where the shade mode is either D3D_GOURAUD or D3D_FLAT.

Example 2-14. Enabling Shading with OpenGL

To enable either flat or Gouraud shading using OpenGL, the following call can be made:

```
glShadeModel(<GLSHADEMODE>
```

Where the shade mode is either GL_SMOOTH, for Gouraud shading, or GL_FLAT for flat shading.

2.2.4.7 Stippled Pattern

The stipple pattern feature of the Intel740™ graphics accelerator is used to set values in a 32x32 pixel matrix to be either 1 or 0, where 0 means that the corresponding portion of the pattern will be rendered as a black pixel. Stippled patterns can be used when the application wants the screen to fade to black by changing the pattern to have more zeros set for each frame rendered.

Example 2-15. Enabling Stippled Patterns with DirectX

To enable stippled pattern for the Intel740™ graphics accelerator using DirectX, do the following:

```
SetRenderState(D3DRENDERSTATE_STIPPLEENABLE, TRUE);
SetRenderState(D3DRENDERSTATE_STIPPLEPATTERN00, 1 OR 0);
    .
    .
SetRenderState(D3DRENDERSTATE_STIPPLEPATTERN31, 1 OR 0);
```

The default value for all of the stipple patterns is 0. When a stippled pattern is enabled and no stipple pattern is set, the result is a black screen.

Example 2-16. Enabling Stippled Patterns with OpenGL

To enable stippled pattern for the Intel740™ graphics accelerator using OpenGL do the following:

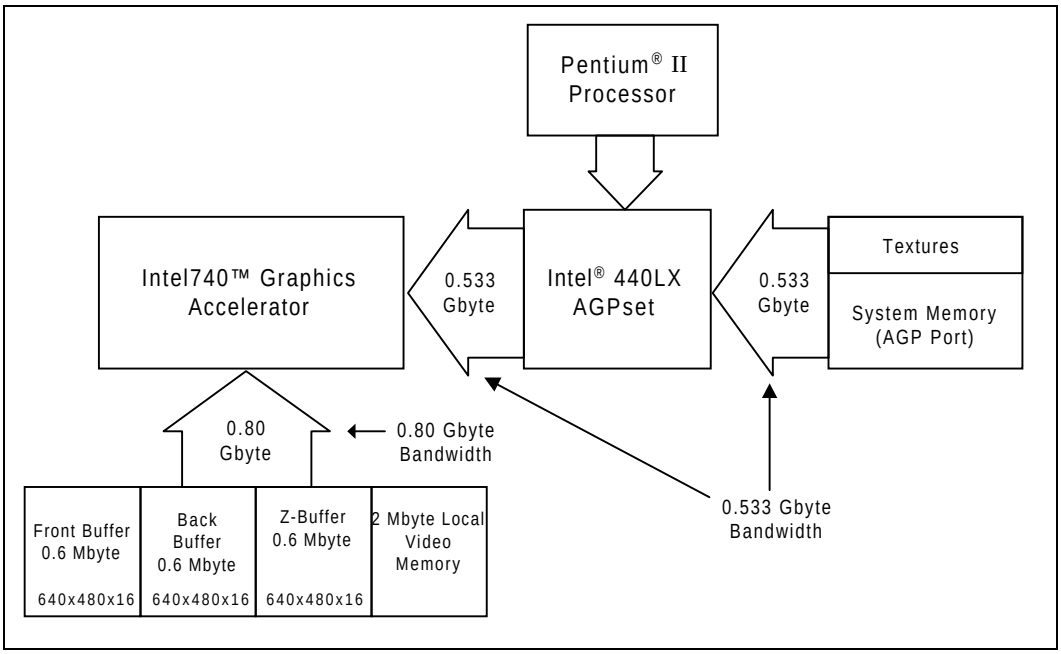
```
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(const Glubyte *StippleMatrix);
```

Where StippleMatrix is a pointer to a 32x32 pixel bitmap interpreted as a mask of 0s and 1s.

2.2.5 Texture Map Attributes

The Intel740™ graphics accelerator allows virtually unlimited texture usage. This is because textures can be stored in the AGP system memory (non-local video memory). The amount of AGP memory available for the application is limited by the amount of system RAM which can be allocated. Therefore, if a system has 32 Mbytes of RAM available, 20 Mbytes could be used for textures. Using AGP for texture memory complements the performance of the Intel740™ graphics accelerator, since textures can be mapped directly from AGP memory to the Intel740™ graphics accelerator without using the CPU. This mapping is done in parallel with the Intel740™ graphics accelerator local video memory transfers for frame buffers. The total bandwidth enabled by the parallel throughput is up to 1.3 Gbytes per second. The Intel740™ graphics accelerator also “tiles” textures in AGP memory to minimize page faults and storage overhead which increases both performance and texture space. Textures can not be put in local video memory.

Figure 2-14. Getting 1.3 Gbytes of Concurrent Throughput with the Intel740™ Graphics Accelerator



There are many ways to manipulate surface textures with the many Intel740™ graphics accelerator Texture Map Attributes. The categories are described in the following subsections:

- “Texture Map Formats” on page 2-26
- “Texture Map Blending” on page 2-29
- “Texture Map Color Keying” on page 2-31
- “Texture Wrapping Formats” on page 2-33
- “Texture Map Filtering” on page 2-34
- “Texture Mipmapping” on page 2-36

2.2.5.1 Texture Map Formats

The Intel740™ graphics accelerator supports up to 16 bits of color in various texture formats. There are three ways to catalog texture types: ARGB, AYUV, or YUV. All the texture formats listed below are supported as either palettized or non-palettized. When the amount of bits per texel in a texture is less than 16, the color information is stored in a palette consisting of 256 16-bit entries. The texture cache is used to store previously accessed texels needed for blending or other purposes, so that additional reads from memory are not needed. The Intel740™ graphics accelerator supports images whose dimensions are a power of two. The dimensions do not have to be square.

DirectX Texture Map Formats supported:

- 1555ARGB
- 0565ARGB (DirectX default for palettized)
- 4444ARGB (DirectX default for palettized with alpha)
- 422YUV (UV is 2's complement) (YUY2 FOURCC)
- 422YUV (UV is excess 128) (YUY2 FOURCC)
- 0555AYUV (texture data compression)
- 1544AYUV (texture data compression)
- Palettized 1, 2, 4, and 8 bit.

OpenGL Texture Map Formats supported:

- RGB5 (0555ARGB)
- RGBA4(4444ARGB)
- RGB5_A1(1555ARGB)

Example 2-17. Creating a Texture Surface with DirectX

The following DirectX example shows how to create a 4444 ARGB texture surface in AGP memory:

First set the pixel format for the 4444 ARGB:

```
DDPIXELFORMAT ddpf;
DDSURFACEDESC ddsd;
ddpf.dwSize = sizeof(ddpf);
ddsd.dwSize = sizeof(ddsd);
ddpf.dwRGBBitCount = 16   //Total number of bits including alpha
ddpf.dwRBitMask = 0x0F00; //Specify the masks for color components
ddpf.dwGBitMask = 0x00F0;
ddpf.dwBBitMask = 0x000F;
ddpf.dwRGBAlphaBitMask = 0xF000;
ddpf.dwFlags = DDPF_RGB;  //specify the pixel format is valid
ddsd.dwFlags = DDSD_PIXELFORMAT;
```

Next set the correct direct draw surface capability flags and creates the surface:

```
IDIRECTDRAW*lpdd;
IDIRECTDRAWSURFACE*lpTextureSurface;
HRESULT ddrval;
ddsd.dwSize = sizeof(ddsd);
ddsd.dwHeight = 128;
ddsd.dwWidth = 128;
ddsd.wFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps = DDSCAPS_TEXTURE | DDSCAPS_ALLOCONLOAD | DDSCAPS_VIDEOMEMORY |
       DDSCAPS_NONLOCALVIDMEM;
ddrval = lpdd->CreateSurface(&ddsd, &lpTextureSurface, NULL);
```

Once a texture surface has been created, a palette and texture can be loaded onto the surface using the DirectDraw sample functions DDLoadPalette and DDReLoadBitmap from the ddutil.cpp file included in the DirectX 5.0 SDK.

IDIRECTDRAWPALETTE *lpDDPal;

```
lpDDPal = DDLoadPalette(lpDD, “MyTexture.bmp”);
ddrval = lpTextureSurface->SetPalette(lpDDPal);
ddrval = DDReLoadBitmap(lpTextureSurface, “MyTexture.bmp”);
```

To enable the texture for rendering, the following state change is made where the texture handle which points to a texture surface is enabled so that a particular texture surface will be rendered:

```
D3DTEXTUREHANDLE HTex;
lpTextureSurface->GetHandle(lpD3Ddevice, &HTex);
SetRenderState(D3DRENDERSTATE_TEXTUREHANDLE, &HTex);
```

The texture handle assigned to the texture surface is enabled.

Example 2-18. Creating a Texture Surface with OpenGL

In OpenGL 1.1, it is recommended to use texture objects. Texture objects are beneficial because they allow the programmer to specify which texture is active with one simple call after these three steps are taken:

1. Generate texture names; a texture name can be any nonzero unsigned integer. The following call should be used when generating a texture name to ensure that a unique texture name is created.

   ```
   glGenTextures(GLsize n, Gluint *TextureName);
   ```

   This call returns a texture object pointed to through textureName. When using an array of texture names, *n* corresponds to the number of unused textures names in the array of texture names.

2. The next step is to bind texture objects to texture data. The following call is used:

   ```
   glBindTexture(GLenum target, Gluint *TextureName);
   ```

   This causes the texture specified by TextureName to become active where target is either GL_TEXTURE_1D, or GL_TEXTURE_2D and TextureName is the same pointer used in glGenTextures.

3. The next step creates the texture surface which will from then on, correspond to the textureName pointer:

   ```
   glTexImage2D(GLenum <TARGET>, GLint <LEVEL>, GLint <INTERNALFORMAT>,
   Glsizei<WIDTH>, GLsize <HEIGHT>, GLint <BORDER>, GLenum <FORMAT>, GLenum
   <TYPE>, GLvoid <PIXELS>);
   ```

   <TARGET> is either GL_TEXTURE_2D, or GL_PROXY_TEXTURE_2D;

   <LEVEL> is 0 or the number of texture resolutions to be used

   <INTERNALFORMAT> is the texture format supported by the Intel740™ graphics accelerator and is GL_RGB5 or GL_RGBA4, or GL_RGB5_A1

   <WIDTH> and <HEIGHT> correspond to the dimensions of the texture; <BORDER> indicates the width of the border which is either 0 (if there is no border) or 1

   <FORMAT> and <TYPE> describe the format and data type of the texture image data

   <PIXELS> is a pointer to the texture image data. This data describes the texture image itself as well as its border.

When put together, creating and enabling a texture surface is done by the following:

```
glEnable(GL_TEXTURE_2D);
glGenTextures(1, &texture_name);
glBindTexture(GL_TEXTURE_2D, texture_name);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16, width, height, 0, GL_RGBA16,
GL_UNSIGNED_BYTE, image_pointer);
```

The variable image_pointer points to the memory location where the image data is currently stored. Subsequent uses of the same image data need only use the glEnable() and the glBindTexture() calls.

2.2.5.2 Texture Map Blending

The Intel740™ graphics accelerator supports texture map blending modes that can be used to modify the pixel color by blending a textured surface with the underlying vertex color.

Example 2-19. Enabling Texture Blending with DirectX

DirectX texture blending modes are shown in Table 2-3. Each mode's behavior depends on whether a texture alpha is provided (RGB or RGBA). The color and alpha equations are given for each case. These equations employ the following definitions:

Cf	intrinsic (flat or Gouraud Interpolated) color of feature
Af	intrinsic (flat or Gouraud Interpolated) alpha of feature
Ct	color from texture data
At	alpha from texture data
Am	lsb of nearest-neighbor alpha from texture data
Co	color output of texture blend function
Ao	alpha output of texture blend function

Some of the modes degenerate to the same function if a texture alpha is not provided.

Table 2-3. DirectX Texture Map Blending Functions

Mode	Texture Mode	Pixel Color	Alpha	D3D Texture Modes (D3DBLEND_)
Decal	RGB	Co = Ct	Ao = Af	DECAL
Decal	RGBA	Co = Ct	Ao = At	
Modulate	RGB	Co = Cf * Ct	Ao = Af	MODULATE
Modulate	RGBA	Co = Cf * Ct	Ao = At	
Decal Alpha	RGB	Co = Ct	Ao = Af	DECALALPHA
Decal Alpha	RGBA	Co = (1-At)*Cf + At*Ct	Ao = Af	
Modulate Alpha	RGB	Co = Cf * Ct	Ao = Af	MODULATEALPHA
Modulate Alpha	RGBA	Co = Cf * Ct	Ao = Af * At	
Decal Mask	RGB	Co = Ct	Ao = Af	DECALMASK
Decal Mask	RGBA	If (Am) Co = Ct Else Co = Cf	Ao = Af	
Modulate Mask	RGB	Co = Cf * Ct	Ao = Af	MODULATEMASK
Modulate Mask	RGBA	If (Am) Co = Cf * Ct Else Co = Cf	Ao = Af	

Each of the DirectX texture blend states is described in detail below:

DECAL

In the Decal state, the output color is the texture color. The output alpha is the feature alpha with an RGB texel format and the texture alpha with an RGBA texel format.

MODULATE

In the Modulate state, the output color is the product of the texture color and the feature color. The output alpha is the feature alpha with an RGB texel format and is the texture alpha with an RGBA texel format.

DECALALPHA

In the Decal Alpha state, the output color is the texture color with an RGB texel format and is a texture alpha blended combination of the feature color and the texture color with an RGBA texel format. The output alpha is the feature alpha.

MODULATEALPHA

In the Modulate Alpha state, the output color is the product of the texture color and the feature color. The output alpha is the feature alpha with an RGB texel format and is the product of the feature alpha and the texture alpha, with an RGBA texel format.

DECALMASK

In the Decal Mask state, the output color is the texture color with an RGB texel format. With an RGBA texel format, the output color is the texture color if the nearest neighbor texel alpha lsb is 1 and is the feature color if the nearest neighbor texel alpha lsb is 0. The output alpha is the feature alpha.

MODULATEMASK

In the Modulate Mask state, the output color is the product of the feature color and the texture color with an RGB texel format. With an RGBA texel format, the output color is the product of the feature color and the texture color if the nearest neighbor texel alpha lsb is 1 and is the feature color if the nearest neighbor texel alpha lsb is 0. The output alpha is the feature alpha.

To use the texture map blending features with DirectX, first obtain a handle to the texture surface to be used for blending:

```
D3DTEXTUREHANDLE HTex;
lpTextureSurface->GetHandle(lpD3Ddevice, HTex);
SetRenderState(D3DRENDERSTATE_TEXTUREHANDLE, &HTex);
SetRenderState(D3DRENDERSTATE_TEXTUREMAPBLEND, <D3DTEXTUREBLEND>);
```

Where the D3DTEXTUREBLEND values are (D3DTBLEND_) DECAL, DECALALPHA, DECALMASK, MODULATE, MODULATEALPHA, MODULATEMASK, or COPY.

Example 2-20. Enabling Texture Blending with OpenGL

Table 2-4 states the texture blend functions for OpenGL which the Intel740™ graphics accelerator supports.

Table 2-4. OpenGL Texture Blend Modes and Equations

Mode	Texture Mode	Pixel Color	Alpha	OpenGL Mode
DECAL	RGB	Co = Ct	Ao = Af	GL_DECAL
DECAL	RGBA	Co = Cf(1-At)+Ct*At	Ao = Af	GL_DECAL
MODULATE	RGB	Co = Cf * Ct	Ao = Af	GL_MODULATE
MODULATE	RGBA	Co = Cf * Ct	Ao = Af*At	GL_MODULATE

To enable texture map blending in OpenGL, the following code is used:

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textureName);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_NEV_MODE , <MODE>);
```

where <MODE> is stated as being either GL_DECAL or GL_MODULATE.

2.2.5.3 Texture Map Color Keying

Color keying is similar to the Hollywood "blue screen" effect whereby a color can be selected in the destination texture through which the source can be made visible. To enable destination color keying, the user selects a color value in the destination surface as the color key and then blits the source texture using the destination color key flag. Another way to use color keying is to make a portion of the source texture inivisible so that only some of the texture is shown on top of the destination surface. Source color keying is a popular way to produce 2D sprites over a 3D background. For source color keying, the user selects a color value in the source texture as the color key value to be made transparent and then performs the blit with that texture using the source color key flag as illustrated in the DirectX source code example below.

Figure 2-15. A Color Keyed Splash

Example 2-21. Enabling Texture Map Color Keying with DirectX

To enable color keying with DirectX, the user fills in a D3DCOLORKEY structure's dwColorSpaceLowValue and dwColorSpaceHighValue with the transparent color's value. For palettized texture, this will be a palette index. For RGBA textures, this will be a 16 bit color value. The rest of the code is as follows:

```
typedef struct D3DCOLORKEY{
    DWORD   dwColorSpaceLowValue;
    DWORD   dwColorSpaceHighValue;
}  DDCOLORKEY;
DDCOLORKEY ColorKeyInfo;
// for non-palettized textures
ColorKeyInfo.dwColorSpaceLowValue = 0x0000;
ColorKeyInfo.dwColorSpaceHighValue =   0x0000;
// for palettized textures
ColorKeyInfo.dwColorSpaceLowValue = 0;
ColorKeyInfo.dwColorSpaceHighValue =  0;
lpTextureSurface->SetColorKey(<dwFlags>, &ColorKeyInfo);
```

Where the <dwFlags> are either, DDCKEY_DESTBLT, DDCKEY_DESTOVERLAY, or DDCKEY_SRCBLT. The SetColorKey function takes as its first parameter a DWORD flag which can specify whether the color key is for a source blit, a destination blit, or a destination overlay.

To enable the color keying, the user needs to set the appropriate render state:

```
SetRenderState(D3DRENDERSTATE_COLORKEYENABLE, TRUE);
```

To actually see color keying, use one of the DirectX Blt functions as shown:

```
lpBackBuffer->BltFast(Xpos, Ypos, lpOffscreenSurface, &Rectangle,
DDBLTFAST_SRCCOLORKEY);
```

2.2.5.4 Texture Wrapping Formats

Applications can specify different texture-wrapping formats for either or both of the U and V directions.

Example 2-22. Enabling Texture Wrapping with DirectX

The Intel740™ graphics accelerator supports the following DirectX texture wrap formats:

Table 2-5. Supported DirectX Texture Wrap Formats

Texture Wrap U	Texture Wrap V	D3DTEXTUREADDRESS
Wrap	Wrap	D3DTADDRESS_WRAP
Mirror	Mirror	D3DTADDRESS_MIRROR
Clamp	Clamp	D3DTADDRESS_CLAMP

WRAP

The wrap mode creates an effect in which the texture map looks like it is repeated over and over in the selected region. In wrap mode, textures appear to be tiled. If either WRAPU or WRAPV is set, the texture is an infinite cylinder with a circumference of 1.0. Texture coordinates greater than 1.0 are valid only in the dimension that is not wrapped.

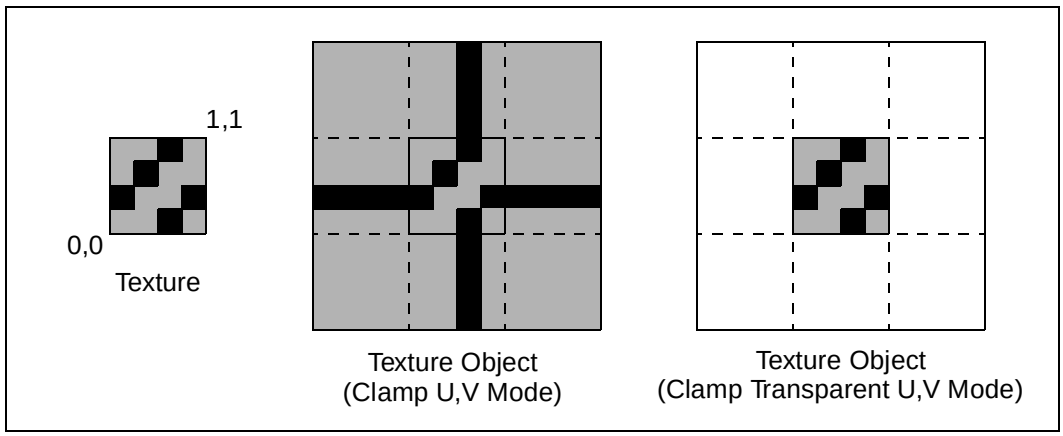
MIRROR

The mirror mode creates an effect where the texture map looks flipped or “mirrored.” It is equivalent to the wrap mode’s “tiling” effect except that the texture is flipped at every integer junction. For instance, between 0 and 1 the texture is normal, then between 1 and 2 the texture is flipped, and between 2 and 3 it is normal, then between 3 and 4 it is flipped, etc.

CLAMP

In clamp mode, the texture coordinates greater than or equal to 1.0 are set to (impasses - 1)/mapsize, and values less than 0.0 are set to 0.0. Figure 2-16 illustrates the effect of the clamp modes. The base texture map is shown, along with two texture mapped objects. Both of these objects have texture coordinates that fall outside of the [0,1] range. The object in the middle illustrates Clamp mode (specified for both U and V), where the texels at the edges are replicated outside the [0,1] range. The object at the right illustrates the same object with Clamp Transparent mode, where pixels with texture coordinates outside the [0, 1] range are not rendered.

Figure 2-16. Texture Clamp Mode



In DirectX, the default texture wrap format is D3DADDRESS_WRAP. To change the texture map format with DirectX API, first set the appropriate texture address type:

```
SetRenderState(D3DRENDERSTATE_TEXTUREADDRESS, <D3DTEXTUREADDRESS>);
```

Where the D3DTEXTUREADDRESS is either D3DTADDRESS_WRAP, D3DTADDRESS_MIRROR, or D3DTADDRESS_CLAMP.

Then enable texture wrapping in either the U or V direction by setting the following:

```
SetRenderState(D3DRENDERSTATE_WRAPU, TRUE);
SetRenderState(D3DRENDERSTATE_WRAPV, TRUE);
```

Example 2-23. Enabling Texture Wrapping with OpenGL

The Intel740™ graphics accelerator supports the following OpenGL texture wrap formats:

Table 2-6. Supported OpenGL Texture Wrap Formats

GL_TEXTURE_WRAP_S	GL_TEXTURE_WRAP_T	VALUE
Clamp	Clamp	GL_CLAMP
Repeat	Repeat	GL_REPEAT

In OpenGL, the texture wrap methods are defined as follows:

CLAMP

Any values greater than 1.0 are set to 1.0, and any values less than 0.0 are set to 0.0. Clamping is useful for applications where you want a single copy of the texture to appear on a large surface. If the surface-texture coordinates range from 0.0 to 10.0 in both directions, one copy of the texture appears in the lower corner of the surface.

REPEAT

Any values outside the range of [0,1] will be repeated in the texture map. With repeating textures, if you have a large texture surface with coordinates from 0.0 to 10.0 in both directions, then 100 copies of the texture will be tiled on the screen.

To enable a texture mapping method, the following calls should be made:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, <WRAP_MODE>);
glTexParameterI(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, <WRAP_MODE>);
```

Where <WRAP_MODE> is either GL_CLAMP or GL_REPEAT.

2.2.5.5 Texture Map Filtering

Texture map filtering enables the user to choose the method the hardware uses to calculate the output pixel color as it corresponds to the texture's texel color at the mapped location. Factors which determine the user's screen pixel color include the distance of the object from the viewer and the size of the texture map in relation to the size of the object. In some applications where texture filtering is not used, a close up object can cause a texture to look blocky because each texel is repeated over a square range of pixels.

The Intel740™ graphics accelerator supports the following texture filtering modes for both DirectX and OpenGL: Nearest, Linear, Mip Nearest, Mip Linear, Linear Mip Nearest and Linear Mip Linear. The Mip modes will be discussed in the Texture Mipmapping section.

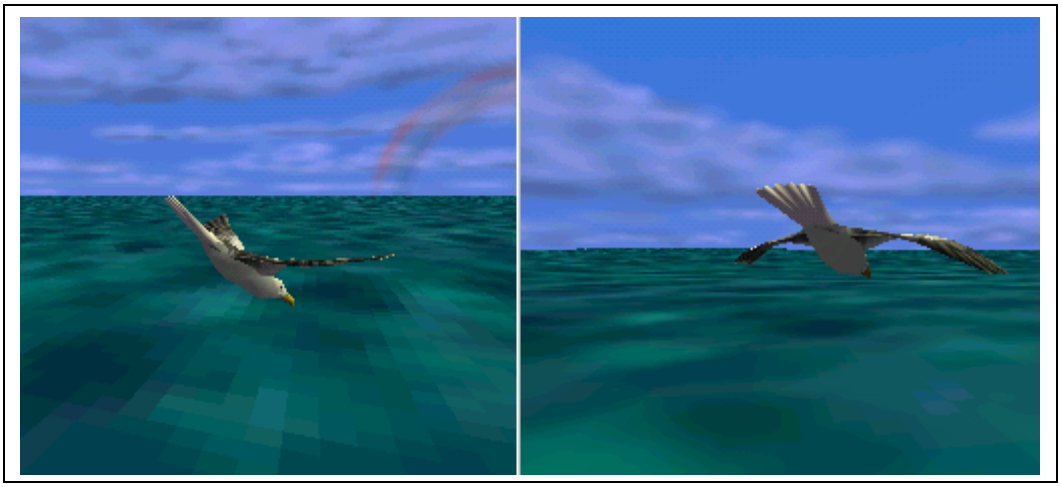
NEAREST

The nearest texture filtering mode is also known as “point filtering.” In this mode, the texel with coordinates nearest to the desired pixel value are used. The output can result in blocky textures as the object becomes larger to the viewer.

LINEAR

The linear texture filtering mode is also known as “bilinear filtering.” In this mode, a weighted average of a 2-by-2 area of texels surrounding the desired pixel is used. The output results in a smoother representation of the texture without blockyness.

Figure 2-17. Point Filtering VS. Bilinear Filtering



Example 2-24. Enabling Texture Map Filtering with DirectX

To enable texture filtering with DirectX, there are two cases which must be addressed. First is when the texture map is minified because the texel is smaller than one pixel. The second case is when the texture map is magnified and a texel is larger than one pixel. To enable texture filtering with DirectX, the following render states must be set:

```
SetRenderState(D3DRENDERSTATE_TEXTUREMIN, <D3DTEXTUREFILTER>);
SetRenderState(D3DRENDERSTATE_TEXTUREMAG, <D3DTEXTUREFILTER>);
```

Where the D3DTEXTUREFILTER can be set to either D3DFILTER_NEAREST, D3DFILTER_LINEAR, D3DFILTER_MIPNEAREST, D3DFILTER_MIPLINEAR, D3DFILTER_LINEARMIPNEAREST, or D3DFILTER_LINEARMIPLINEAR.

Example 2-25. Enabling Texture Map Filtering with OpenGL

To enable texture filtering with OpenGL, the following calls are made:

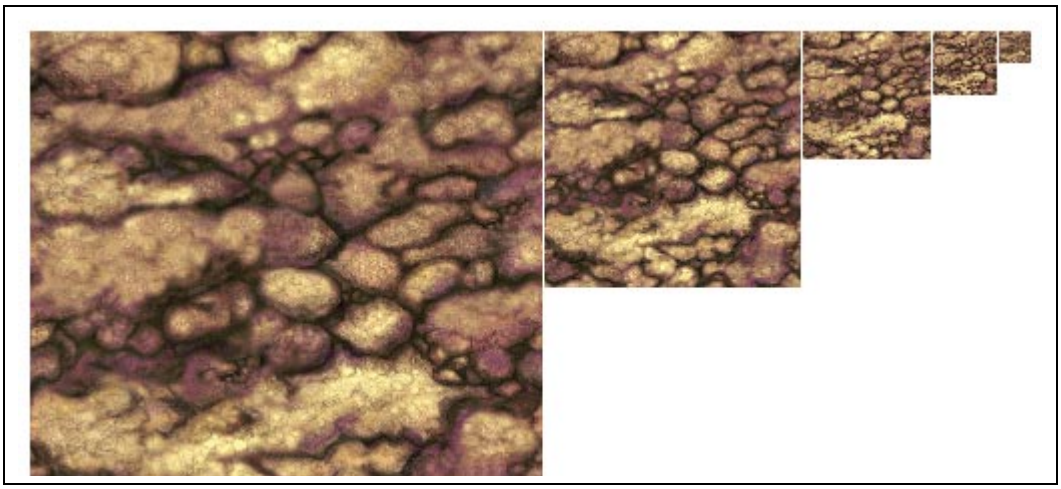
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, <FILTER_MODE>);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, <FILTER_MODE>);
```

Where FILTER_MODE is either GL_NEAREST, GL_LINEAR, GL_MIPMAP_NEAREST, GL_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, or GL_LINEAR_MIPMAP_LINEAR.

2.2.5.6 Texture Mipmapping

Because textured objects can be viewed at different distances from the viewer in 3D space, it is possible for a texture object to become smaller than the texture image. This occurrence will cause the texture map to be under-sampled during rasterization. As a result, the texture mapping may display artifacts or “noise.” The purpose of trilinear interpolating and mipmapping is to minimize this effect. With mipmapping, software provides a series of pre-filtered texture maps of decreasing resolutions, called “mipmaps” and stores them in memory. When a 3D object is larger because of its close proximity to the viewer, a corresponding texture map is used. As the object moves farther away from the viewer, the Intel740™ graphics accelerator determines which mipmap to use and switches to a smaller texture size.

Figure 2-18. An Example of Five Levels of Mipmapped Texture



Intel740™ graphics accelerator supports 11 mipmaps ranging from 1024 x 1024 down to a 1 x 1 texel map. Each successive level has 1/2 the resolution of the previous level in the U and V directions until a 1x1 texture is reached. Both dimensions of the mipmap must be a power of 2 although they do not have to be square. Four forms of mipmap texture filtering that can be selected in either DirectX or OpenGL are:

MIP NEAREST

Similar to the texture filtering Nearest form except that Mip Nearest uses the appropriate mipmap for texel selection.

MIP LINEAR

Similar to the texture filtering Linear form except that Mip Linear uses the appropriate mipmap for texel selection.

LINEAR MIP NEAREST

The two closest mipmap levels are chosen and then a linear blend is used between point filtered samples of each level.

LINEAR MIP LINEAR

The two closest mipmap levels are chosen and then combined using a bilinear filter.

Example 2-26. Mipmap Enabling with DirectX

To enable texture mipmapping using DirectX, a mipmapped surface must first be created and loaded with the appropriate texture maps. To do this with DirectX Immediate Mode, specify that the surface is a TEXTURE surface and also a MIPMAP surface. The user can specify the mipmap count, but this is not necessary. When the "CreateSurface" call is made, DirectX generates all the levels on its own, down to 1x1.

Start by creating the mipmap surfaces:

```
HRESULT                 ddres;
DDSURFACEDESC           ddsd;
LPDIRECTDRAWSURFACE3   lpDDMipMap;
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_MIPMAPCOUNT; ddsd.dwMipMapCount = 5;
ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE | DDSCAPS_MIPMAP | DDSCAPS_COMPLEX
                  | DDSCAPS_VIDEOMEMORY | DDSCAPS_NONLOCALVIDMEM;
ddsd.dwWidth = 256;
ddsd.dwHeight = 256;
```

Then call the CreateSurface function to build the mipmap chain of surfaces:

```
ddres = lpDD->CreateSurface(&ddsd, &lpDDMipMap);
```

Now five subsequent mipmapped surfaces have been created. The next step is to load an image onto each surface. This can be done by traversing the surfaces with the DirectX GetAttachedSurface call and then copying a bitmap which has already been loaded to the current mipmap level surface using the DDCopyBitmap function. See the DirectX SDK manuals and on-line help for more in-depth information.

Finally, enable the mipmap filtering mode by setting the following render state in DirectX:

```
SetRenderState(D3DRENDERSTATE_TEXTUREMIN, <D3DTEXTUREFILTER>);
SetRenderState(D3DRENDERSTATE_TEXTUREMAG, <D3DTEXTUREFILTER>);
```

Where D3DTEXTUREFILTER is D3DFILTER_MIPNEAREST, D3DFILTER_MIPLINEAR, D3DFILTER_LINEARMIPNEAREST, or D3DFILTER_LINEARMIPLINEAR.

Example 2-27. Enabling Mipmapping with OpenGL

OpenGL has a function which generates all the mipmaps from the dimensions of the mipmap specified down to 1x1. The dimensions of the mipmap can be any power of 2. The following call is used:

```
gluBuild2DMipmaps(GLenum target, Glint components, Glint width, Glint height,
Glenum format, Glenum type, void *data);
```

This function is like the glTexImage2D() which creates a texture map surface as mentioned in the section above.

Then, enable either the mip-nearest or mip-linear filtering mode with the following function call:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, <FILTER_MODE>);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, <FILTER_MODE>);
```

Where FILTER_MODE is GL_MIPMAP_NEAREST or GL_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, or GL_LINEAR_MIPMAP_LINEAR.

2.2.6 Drawing Formats

The Intel740™ graphics accelerator supports the following Drawing Formats:

Solid	The output to the screen is a triangle, either solid color or patterned by a texture map.
Wire-frame	The output to the screen is a line drawing, either solid color or patterned by a texture map.

Example 2-28. Enabling Drawing Formats with DirectX

To enable drawing formats with DirectX, the following render state call is used:

```
SetRenderState(D3DRENDERSTATE_FILLMODE, <D3DFILLMODE>);
```

Where D3DFILLMODE is either D3DFILL_WIREFRAME or D3DFILL_SOLID.

Example 2-29. Enabling Drawing Formats with OpenGL

To enable drawing formats with OpenGL, the following call is made:

```
glPolygonMode(<FACE>, <MODE>);
```

where FACE is GL_FRONT_AND_BACK, GL_FRONT or GL_BACK and MODE is either GL_LINE, or GL_FILL.

2.2.7 Buffers

The Intel740™ graphics accelerator supports many buffer types including:

- A back buffer, which can be placed in local video memory
- A front buffer, which should be placed in local video memory
- A Z-buffer, which must be placed in local video memory

The Intel740™ graphics accelerator also supports two back buffer surfaces needed for triple buffering.

In OpenGL, the buffers are created by selecting the proper pixel format. The pixel formats and the corresponding buffers they create are as follows:

Table 2-7. Pixel Formats and Buffers

Visual ID[1]	Frame Buffer Format	Double Buffer	Depth Buffer Size (bits)	Stencil Buffer Size (bits)	Accumulation Buffer Size (bits)
1	R5_G6_B5	No	0	0	0
1	R5_G6_B5	Yes	0	0	0
3	R5_G6_B5	No	16	0	0
4	R5_G6_B5	Yes	16	0	0
5[2]	R5_G6_B5	Yes	16	8	64

NOTES:
1. Depending on the way your display adapter is configured, the actual Visual ID may differ.
2. Supported only with the OpenGL ICD (Installable Client Driver)

When creating buffers with the DirectX API, the user uses the “CreateSurface” call and sets appropriate DDSD flags and capabilities.

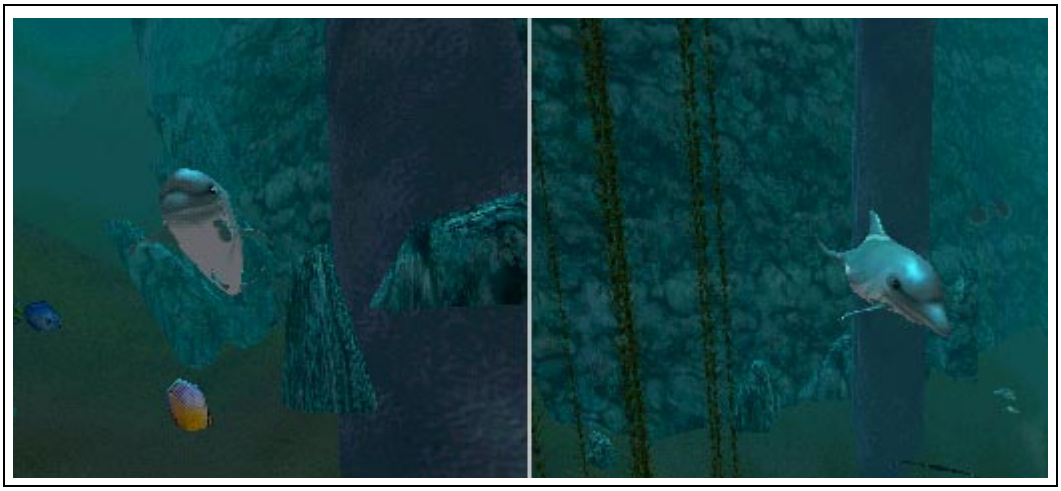
2.2.7.1 Double and Triple Buffering

Intel740™ graphics accelerator permits the use of both double and triple buffering, where one buffer is the primary buffer used for display and one or two are the back buffer(s) used for rendering. With double buffering, an application typically constructs a scene in the back buffer while the front buffer is being displayed. With triple buffering, a flipping chain of buffers is used which gives added buffering between drawing to the back buffer and rendering which can help increase performance. For double buffering, when the scene in the back buffer is complete and it is time to display, the application flips the two buffers or rather, switches the roles of the two buffers so that the drawn-to buffer becomes the rendering buffer and vice versa. In the case of triple buffering, when flipping of the buffers is performed, the application makes the second to last drawn-to buffer the rendering (primary) buffer and draws to the last buffer used for rendering.

2.2.7.2 Z-Buffering

The Z-buffer contains 16 bits of depth information for each pixel in the display buffer. The use of the Z-buffer is optional. Figure 2-19 shows the use of the Z-buffer.

Figure 2-19. Z-Buffering Off vs. Z-Buffering On



When enabled, the Z-buffer function performs a depth compare between the pixel Z (known as source Z or ZS) and the Z value read from the Z-buffer at the current pixel address (known as destination Z or ZD). If the test is not enabled, it is assumed the Z test always passes. The Z value is only written to the Z-buffer when the results of the Z test are true. It is always necessary to clear the Z-buffer before each new frame is drawn.

The Intel740™ graphics accelerator uses a logarithmic method for Z-buffering. The logarithmic approach makes those objects closer to the viewer look better than does the linear approach.

Example 2-30. Enabling Z-Buffering with DirectX

```
To Create a Z-buffer with DirectX the following surface must be created:
DDSURFACEDESC ddsd;
IDIRECTDRAW*lpdd;
IDIRECTDRAWSURFACE*lpZSurface;
HRESULT ddrval;
ddsd.dwSize = sizeof(ddsd);
ddsd.dwHeight = window_height;
ddsd.dwWidth = window_width;
ddsd.dwZBufferBitDepth = 16;
ddsd.wFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH | DDSD_ZBUFFERBITDEPTH;
ddsd.ddsCaps = DDSCAPS_ZBUFFER | DDSCAPS_VIDEOMEMORY | DDSCAPS_LOCALVIDMEM;
ddrval = lpdd->CreateSurface(&ddsd, &lpZSurface, NULL);
```

To enable Z-buffering with DirectX, the following render states must be set:

```
SetRenderState(D3DRENDERSTATE_ZENABLE, TRUE);
SetRenderState(D3DRENDERSTATE_ZWRITEENABLE, TRUE);
SetRenderState(D3DRENDERSTATE_ZFUNC, <D3DCMPFUNC>);
```

D3DCMPFUNC is D3DCMP_NEVER, D3DCMP_LESS, D3DCMP_EQUAL, D3DCMP_GREATEREQUAL, D3DCMP_LESSEQUAL, D3DCMP_GREATER, D3DCMP_NOTEQUAL, or D3DCMP_ALWAYS.

The application also must clear the Z-Buffer using the following DirectX function call:

```
lpZSurface->Blt(lpDestRect, lpDDSrcSurface,lpSrcRec,  DDBLT_DEPTHFILL,
dwFillDepth);
```

Example 2-31. Enabling Z-Buffering with OpenGL

To enable Z-Buffering with OpenGL, the following code is used:

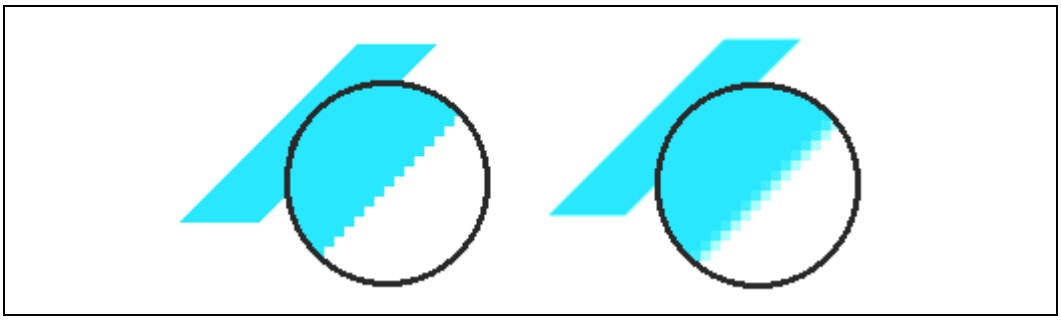
```
glEnable(GL_DEPTH_TEST);
glDepthFunc(<FUNCTION>);
```

FUNCTION is GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or GL_NOTEQUAL.

2.2.8 Antialiasing

Antialiasing will blend the edges of objects so that they appear to be smooth rather than jagged due to the amount of pixel resolution on the screen. It is recommended to enable antialiasing for approximately 20% of the geometry where it will count most as antialiasing does cause a minimal performance decrease. The best way to use antialiasing is to render everything not antialiased first, and then to render the last 20% with antialiasing enabled. Z buffering should always be enabled when using the Intel740 chip so that polygon sorting is not required of the user and to ensure the highest rate of 3D acceleration.

Figure 2-20. Effects of Antialiasing



Example 2-32. Enabling Antialiasing with DirectX

To enable antialiasing with DirectX, the user needs to have Z buffering enabled, Z write enabled, and also a Z function should be defined. Sorting of polygons is not required, although if antialiasing a portion of the scene, that portion should be rendered last. Both the SORTDEPENDENT and SORTINDEPENDENT methods are supported; however, they will both produce the same results and they will both take the same amount of time. Neither method requires that the user pre sort their polygons. Alphablending needs to be disabled when using antialiasing with the Intel740 chip for antialiasing to work.

To enable antialiasing with DirectX, the following render state is enabled:

```
SetRenderState(D3DRENDERSTATE_ANTIALIAS, SORTDEPENDENT);
```

When using execute buffers, an edge flag can be set to enable edge antialiasing.

Example 2-33. Enabling Antialiasing with OpenGL

To enable antialiasing with OpenGL, the user needs to have a blending method enabled and a blending function selected depending on how their application is created. It is recommended that the user does not sort their polygons, but relies on the Intel740 chip's Z buffering for more hardware acceleration. The following code can be used to enable antialiasing when Z buffering is enabled:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_LINE_SMOOTH);
```

2.2.9 Back Face Culling

One of the stages in the 3D Pipeline which can be performed in either the software geometry stage or in the hardware rendering stage is that of back face culling which consists of the removal of surfaces of 3D objects which cannot be seen from the user's viewpoint. The Intel740™ graphics accelerator supports back face culling. Because every surface has a surface normal which is a vector perpendicular to its surface, the normals of each surface can be tested to see if they point backwards away from the viewer. Back face culling saves processing time since culled surfaces will not need to be rendered. When using color alpha blending, be sure to disable back face culling because alpha blending looks better when the back facing polygons are also rendered and are visible through the translucent alpha blended portions.